Java is Platform Independent

By Dr. Subrat Kumar Nayak

Associate Professor,

Department of CSE, FET, S'O'A (Deemed to be) University.

General Procedure

- What is a Platform?

Platform is combination of processor and OS(operating system). In general we can say the hardware or software component in which programs run.

- How program get executed?

When you write program in C/C++ and when you compile it, it is directly converted into machine readable language(.exe).

- Two types of codes get generated after compilation need to be focused.

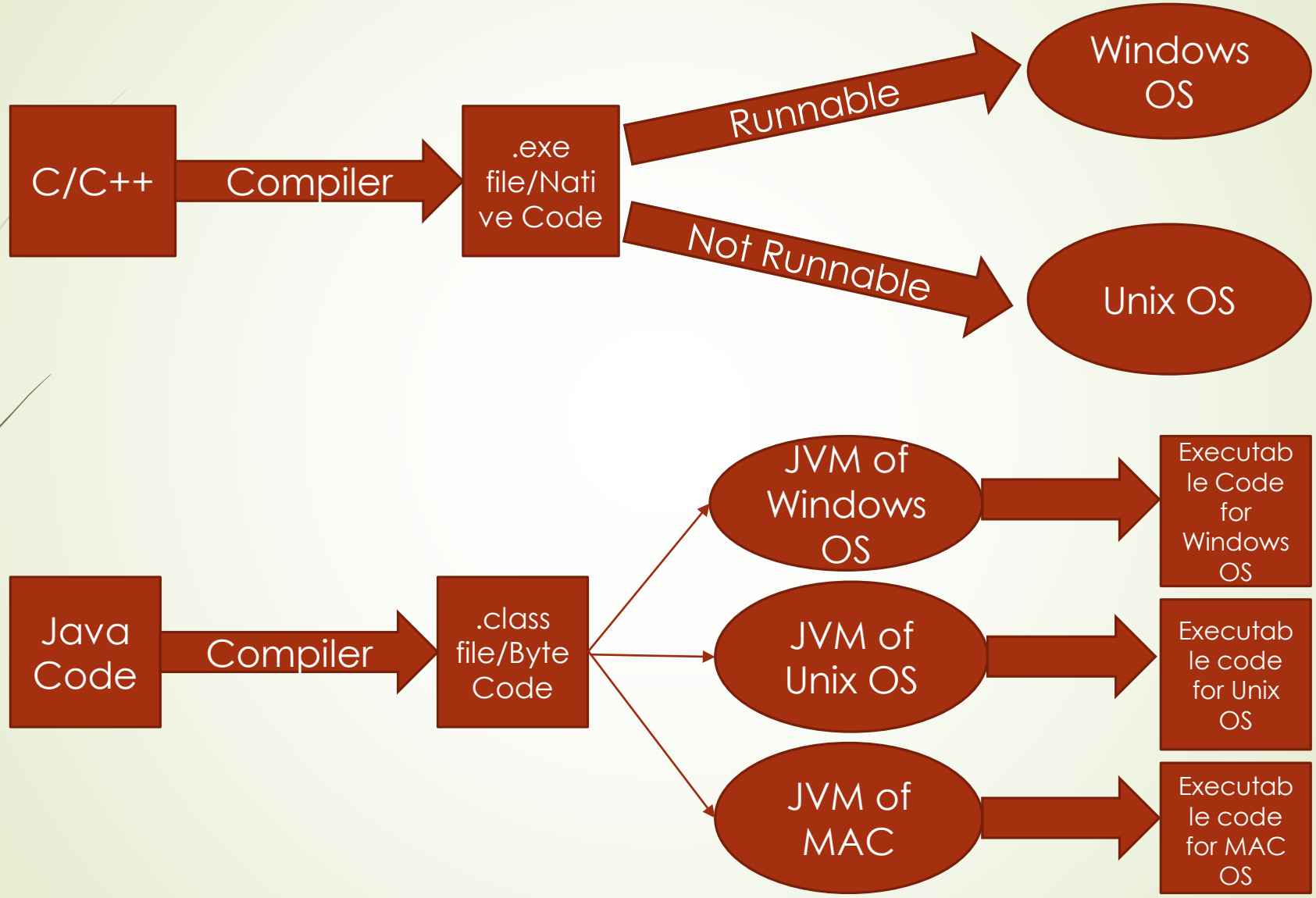
1. Native code

   Native code is similar to machine code i.e codes that is understood by machine. Native codes are specific to platform i.e, Native code generated by program for Windows OS is different from Native code generated for the same program for Unix OS.

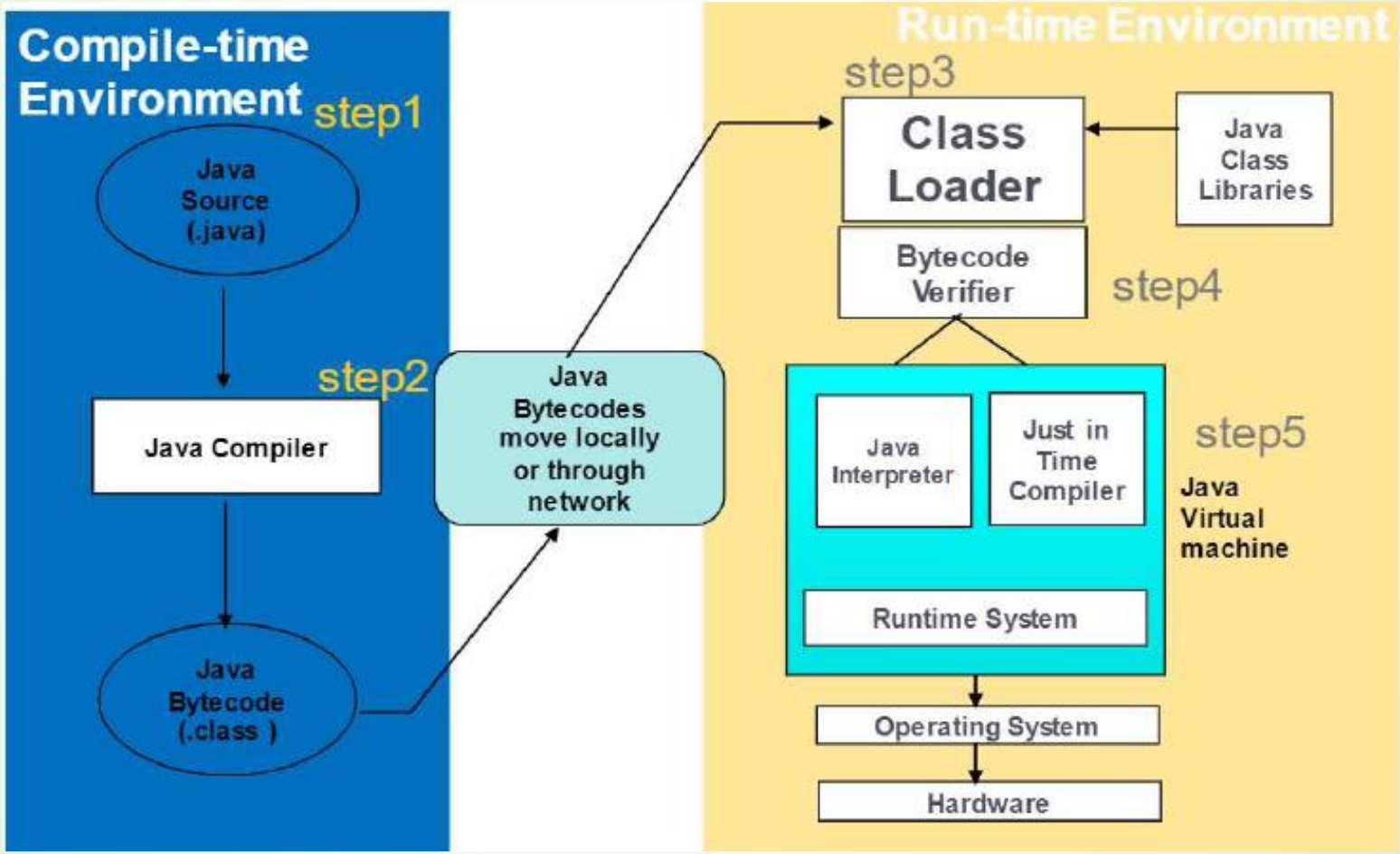
1. Byte code

   Byte codes are nothing but intermediate codes generated after compilation and it is not the executable code like Native code. The Byte code requires a virtual machine to execute in machine. Byte codes generated by one platform can be executed in another platform also.

- Java compiler converts the source code to .class file (byte code).
- Whether JVM is platform independent?


C/C++
Compiler
.exe file/Native Code
Runnable
Windows OS
Not Runnable
Unix OS
Java Code
Compiler
.class file/Byte Code
JVM of Windows OS
JVM of Unix OS
JVM of MAC
Executable Code for Windows OS
Executable code for Unix OS
Executable code for MAC OS

Java Architecture



Java main()

- `public static void main(String[] args)` is the most important java method.

`public`: This is access modifier of `main()`. It has to be public so that java runtime can access this method.

`static`: `main()` function need to be static so that JVM can load the class to memory and call the main method.

`void`: Java `main()` does not return anything.

`String[] args`: Java `main()` accepts a single argument of type String array. This is also called as java command line arguments.

End of Class

Interacting with the Environment, using java.lang.System class

By Dr. Subrat Kumar Nayak

Associate Professor,

Department of CSE, FET, S'O'A (Deemed to be) University.

Introduction

- This chapter describes how your Java program can deal with its immediate surroundings, with what we call the runtime environment.
- In other sense, everything we do in a Java program using almost any Java API involves the environment.
- Here, the focus more narrowly on things that directly surround your program.
- This also talks about the **System** class that will help us to know a lot about the system.
- Many operating systems use *environment variables* to pass configuration information to applications. Like properties in the Java platform, environment variables are **key/value pairs**, where both the key and the value are strings.
- The conventions for setting and using environment variables vary between operating systems, and also between command line interpreters.

Getting Environment Variables

- Here the intention is to get the value of "environment variables" from within your Java program.
- Environment variables are commonly used for customizing an individual computer user's runtime environment.
- How to read environment variable using Java API?
- System class provides two methods
- ✓ `System.getenv(String name)` - which returns specific variable value
- ✓ `System.getenv()` which returns all environment variables values.

Getting Information from System Properties

- Get information from the system properties.
- A property is just a name and value pair stored in a java.util.Properties object.
- In Properties, we examined **the way an application can use Properties objects to maintain its configuration**. The Java platform itself uses a Properties object to maintain its own configuration. The **System** class maintains a **Properties object** that describes the **configuration of the current working environment**.
- System properties include **information about the current user, the current version of the Java runtime, and the character used to separate components of a file path name.**
- The following table describes some of the most important system properties

Key	Meaning
`"file.separator"`	Character that separates components of a file path. This is "/" on UNIX and "\" on Windows.
`"java.class.path"`	Path used to find directories and JAR archives containing class files. Elements of the class path are separated by a platform-specific character specified in the **path.separator** property.
`"java.home"`	Installation directory for Java Runtime Environment (JRE)
`"java.vendor"`	JRE vendor name

Key	Meaning
"java.vendor.url"	JRE vendor URL
"java.version"	JRE version number
"line.separator"	Sequence used by operating system to separate lines in text files
"os.arch"	Operating system architecture
"os.name"	Operating system name
"os.version"	Operating system version
"path.separator"	Path separator character used in java.class.path
"user.dir"	User working directory
"user.home"	User home directory
"user.name"	User account name

Continue…(Reading System Properties)

- The **System** class has two methods used to read system properties:

  `getProperty` and `getProperties`

- To retrieve one system-provided property, use `System.getProperty()`. If you want them all, use `System.getProperties()`.
- Example: `System.getProperty("path.separator");`

Learning About the Current JDK Release

- You need to write code that looks at the current JDK release
- Use `System.getProperty()` with an argument of `java.specification.version`.
- Solution:
  `System.out.println(System.getProperty("java.specification.version"))`

Dealing with Operating System–Dependent Variations

- To write code that adapts to the underlying operating system.
- Though Java is designed to be portable, some things aren't.
- Such as file separetor (/) for unix and (\) for dos or windows.
- You can use System.Properties to find out the operating system, and various features in the File class to find out some platform-dependent features.
- Example:

  use System . getProperty (" file . separator ")

  or

  use String ret = java .io. File . separator ;

End of Chapter

String and things

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

String and things

- In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

  char[] ch={'I','T','E','R'};

  String s=**new** String(ch);

- The above one can also be written as follows.

  String s="ITER";

- Java **String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
- The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.
- The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.
- The Java String is **immutable** which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

String ?

- Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to Create a String object?

- There are two ways to create String object:
  - By string literal
  - By new keyword

By String literal

- Java String **literal** is created by using double quotes. For Example:

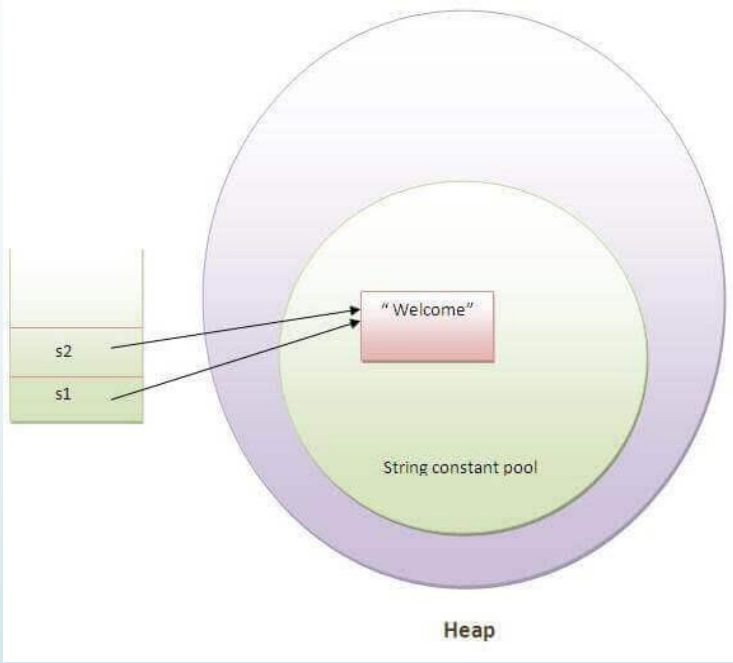
```
String s="welcome";
```

- Each time you create a string literal, the JVM checks the **"string constant pool"** first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```

String…

- In the previous example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.



Why Java uses the concept of String literal?

- To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

By new keyword

- String s=**new** String("Welcome");//creates two objects and one reference variable
- In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

String...

```
public class StringExample{
public static void main(String args[]){
    String s1="java";//creating string by java string literal
    char ch[]={'s','t','r','i','n','g','s'};
    String s2=new String(ch);//converting char array to string
    String s3=new String("example");//creating java string by new keyword
    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s3);
}
}
```

Output:

```
java
strings
example
```

String is Immutable

- In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.
- Let's try to understand the immutability concept by the example given below:

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Siksha";
   s.concat(" O Anusandhan");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects

 }
}
```

Output: Siksha

String is Immutable

- Here Siksha is not changed but a new object is created with Siksha O Anusndhan. That is why string is known as immutable.
- But if we explicitely assign it to the reference variable, it will refer to " Siksha o Anusandhan" object.For example:

```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Siksha";
   s=s.concat(" O Anusandhan");
   System.out.println(s);
 }
}
```

```
Output: Siksha O Anusandhan
```

- In such case, s points to the "Siksha o Anusandhan". Please notice that still Siksha object is not modified.

End of Session

String and things

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

String and things

Taking Strings Apart with Substrings

- Problem: You want to break a string apart into substrings by position.
- Solution: Use the String object's **substring()** method.

substring()

- A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.

Syntax 1:

```
public String substring(int begIndex)
```

Parameters :

begIndex : the begin index, inclusive.

Return Value : The specified substring.

This method returns new String object containing the substring of the given string from specified startIndex (inclusive).

String and things

Syntax 2:

public String substring(int begIndex, int endIndex)

Parameters :

beginIndex : the begin index, inclusive.

endIndex : the end index, exclusive.

Return Value : The specified substring.

This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

Q: Write a program to find sub string of a string from a start index (eg. 5).

Q: Write a program to find sub string of a string from a start index to end index (eg. 2,5).

Q: Write a program that read two strings from user and concat the sub string from 0 to 5 index of first string with the 0 to 7 index of the second string and print the resultant string.

String and things

indexof()

- The **java string indexOf()** method returns index of given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

Syntax for Signature:

There are 4 types of Signatures.

- int indexOf(int ch)

returns **index position** for the given char value

- int indexOf(int ch, int fromIndex)

returns **index position** for the given char value and from **index**

- int indexOf(String substring)

returns **index position** for the given **substring**

- int indexOf(String substring, int fromIndex)

returns **index position** for the given **substring** and **from index**

String and things

```
public ublic class IndexOfExample{
public static void main(String args[]){
    String s1="this is index of example";
    //passing substring
    int index1=s1.indexOf("is");//returns the index of is substring
    int index2=s1.indexOf("index");//returns the index of index substring
    System.out.println(index1+"  "+index2);//2 8
    //passing substring with from index
    int index3=s1.indexOf("is",4);//returns the index of is substring after 4th index
    System.out.println(index3);//5 i.e. the index of another is
    //passing char value
    int index4=s1.indexOf('s');//returns the index of s char value
    System.out.println(index4);//3
}}
```

Output: 2 8

5

3

String and things

lastIndexOf()

The **java string lastIndexOf()** method returns last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

Syntax for Signature

- 4 types

(1) int lastIndexOf(int ch)

returns last index position for the given char value

(2) int lastIndexOf(int ch, int fromIndex)

returns last index position for the given char value and from index

(3) int lastIndexOf(String substring)

returns last index position for the given substring

(4) int lastIndexOf(String substring, int fromIndex)

returns last index position for the given substring and from index

String and things

```
public class LastIndexOfExample{
  public static void main(String args[]){
    String s1="this is index of example";//there are 2 's' characters in this sentence

    int index1=s1.lastIndexOf('s');//returns last index of 's' char value
    System.out.println(index1);//6
}}
```

Output: 6

```
public class LastIndexOfExample2 {
    public static void main(String[] args) {
        String str = "This is index of example";
        int index = str.lastIndexOf('s',5);
        System.out.println(index);
    }
}
```

Output: 3

String and things

```
public class LastIndexOfExample3 {
    public static void main(String[] args) {
        String str = "This is last index of example";
        int index = str.lastIndexOf("of");
        System.out.println(index);
    }
}
```

Output:19

```
public class LastIndexOfExample4 {
    public static void main(String[] args) {
        String str = "This is last index of example";
        int index = str.lastIndexOf("of", 25);
        System.out.println(index);
        index = str.lastIndexOf("of", 10);
        System.out.println(index); // -1, if not found
    }
}
```

Output: 19

-1

End of Session

String and things

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

String and things

length()

The **java string length()** method length of the string. It returns count of total number of characters.

Signature:

- `public int length()`

Example:

```
public class LengthExample{
public static void main(String args[]){
String s1=“java";
String s2="python";
System.out.println("string length is: "+s1.length());//4 is the length of java s
tring
System.out.println("string length is: "+s2.length());//6 is the length of python
 string
}}
```

Output:

```
string length is: 4
string length is: 6
```

String and things

equals()

- The **java string equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.
- The String equals() method overrides the equals() method of Object class.

Signature:

- **public boolean** equals(Object anotherObject)

Example:

```
public class EqualsExample{
public static void main(String args[]){
        String s1="java";
        String s2="java";
        String s3="JAVA";
        String s4="python";
        System.out.println(s1.equals(s2));//true because content and case is same
        System.out.println(s1.equals(s3));//false because case is not same
        System.out.println(s1.equals(s4));//false because content is not same
}}
```

String and things

Q write a program to differentiate between == and equals() method in java.

compareTo()

- The **java string compareTo()** method compares the given string with current string lexicographically. It returns positive number, negative number or 0.
- It compares strings on the basis of Unicode value of each character in the strings.
- If first string is lexicographically greater than second string, it returns positive number (difference of character value). If first string is less than second string lexicographically, it returns negative number and if first string is lexicographically equal to second string, it returns 0.
- **if** s1 > s2, it returns positive number
- **if** s1 < s2, it returns negative number
- **if** s1 == s2, it returns 0

Signature:

- **public int** compareTo(String anotherString)

Q: Write a program to demonstrate the functionality of compareTo() function.

String and things

trim()

- The **java string trim()** method eliminates leading and trailing spaces. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.
- The string trim() method doesn't omits middle spaces

Signature

- **public** String trim()

Example:

```
public class StringTrimExample{
public static void main(String args[]){
    String s1="  hello string   ";
    System.out.println(s1+"java");//without trim()
    System.out.println(s1.trim()+"java");//with trim()
}}
```

Output:

```
   hello string    java
hello stringjava
```

End of Session

String and things

(Breaking Strings into Words)

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Breaking Strings into Word

Problem: You need to take a string apart into words or tokens.

Solutions: To accomplish this, construct a StringTokenizer around your string and call its methods hasMoreTokens() and nextToken(). Another way is to use the String `split()` function for this purpose.

split()

- The java string split() method splits this string against given regular expression and returns an array of string.

Signature

- There are two types of signature for this function

(1) Public String[] split(String regex)

(2) Public String[] split(String regex, int limit)

- Regex: regular expression to be applied on string.
- Limit: limit for the number of stings in the array. If it is zero, it will return all the strings matching regex.

Breaking Strings into Word

Example:

```
public class SplitExample{
public static void main(String args[]){
    String s1="java string split method by ITER";
    String[] words=s1.split("\\s");//splits the string based on whitespace
    //using java foreach loop to print elements of string array
    for(String w:words){
        System.out.println(w);
    }
}}
```

Output:

java
string
split
method
by
ITER

Breaking Strings into Word

```
public class SplitExample2{
    public static void main(String args[]){
        String s1="welcome to split world";
        System.out.println("returning words:");
        for(String w:s1.split("\\s",0)){
            System.out.println(w);
        }
        System.out.println("returning words:");
        for(String w:s1.split("\\s",1)){
            System.out.println(w);
        }
        System.out.println("returning words:");
        for(String w:s1.split("\\s",2)){
            System.out.println(w);
        }
}}
```

Output:
returning words:
welcome
to
split
world
returning words:
welcome to split world
returning words:
welcome
to split world

Breaking Strings into Word

StringTokenizer in Java

- The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

- **StringTokenizer(String str)**

It creates StringTokenizer with specified string. Considers default delimeters like new line, tab and space etc.

- **StringTokenizer(String str, String delim)**

It creates StringTokenizer with specified string and delimeter.

- **StringTokenizer(String str, String delim, boolean returnValue)**

It creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Breaking Strings into Word

Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

- boolean hasMoreTokens()

It checks if there is more tokens available.

- String nextToken()

It returns the next token from the StringTokenizer object.

- String nextToken(String delim)

It returns the next token based on the delimeter.

- boolean hasMoreElements()

It same as hasMoreTokens() method.

- Object nextElement()

It same as nextToken() but its return type is Object

- int countTokens()

returns the total number of tokens.

Breaking Strings into Word

```
import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        System.out.println("Using Constructor 1 - ");
        StringTokenizer st1 =
             new StringTokenizer("Hello Subrat How are you", " ");
        while (st1.hasMoreTokens())
            System.out.println(st1.nextToken());

        System.out.println("Using Constructor 2 - ");
        StringTokenizer st2 =
             new StringTokenizer("JAVA : Code : String", " :");
        while (st2.hasMoreTokens())
            System.out.println(st2.nextToken());

        System.out.println("Using Constructor 3 - ");
        StringTokenizer st3 =
             new StringTokenizer("JAVA : Code : String", " :",  true);
        while (st3.hasMoreTokens())
            System.out.println(st3.nextToken());
    }
}
```

Output:

```
Using Constructor 1 –
Hello
Subrat
How
are
you
Using Constructor 2 –
JAVA
Code
String
Using Constructor 3 –
JAVA

:

Code

:

String
```

End of Session

String and things

(StringBuffer & StringBuilder)

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Java StringBuffer class

- Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.
- Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Constructors:

- StringBuffer()

creates an empty string buffer with the initial capacity of 16

- StringBuffer(String str)

creates a string buffer with the specified string

- StringBuffer(int capacity)

creates an empty string buffer with the specified capacity as length

Java StringBuffer class

Some Important Functions:

append(String s)

- This is used to append the specified string with this string.
- The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

Signature:

- public StringBuffer append(String s)

Example:

```
class StringBufferExample{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello ");
    sb.append("Java");//now original string is changed
    System.out.println(sb);
}}
output:
Hello Java
```

Java StringBuffer class

insert(int offset, String s)

- This is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.

Signature:

- public StringBuffer insert(int offset, String s)

Example:

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);
}}
```

Output:

```
HJavaello
```

Java StringBuffer class

replace()

- This is used to replace the string from specified startIndex and endIndex.

Signature:

- public StringBuffer replace(int startIndex, int endIndex, String str)

Example:

```
class StringBufferExample3{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello");
    sb.replace(1,3,"Java");
    System.out.println(sb);
}}
```

Output:

```
HJavalo
```

Java StringBuffer class

delete()

- This is used to delete the string from specified startIndex and endIndex.

Signature:

- public StringBuffer delete(int startIndex, int endIndex)

Example:

```
class StringBufferExample4{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello");
    sb.delete(1,3);
    System.out.println(sb);
}}
```

Output:

```
Hlo
```

Java StringBuffer class

reverse()

- This is used to reverse the string.

Signature

- public StringBuffer reverse()

Example

```
class StringBufferExample5{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello");
    sb.reverse();
    System.out.println(sb);
}}
```

Output:

```
olleH
```

Java StringBuffer class

capacity()

- The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.
- **Signature:**

public int capacity()

Example:

```
class StringBufferExample6{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer();
    System.out.println(sb.capacity());//default 16
    sb.append("Hello");
    System.out.println(sb.capacity());//now 16
    sb.append("java is my favourite language");
    System.out.println(sb.capacity());
    //now (16*2)+2=34 i.e (oldcapacity*2)+2
}}
```

Java StringBuffer class

Some more functions

- public char charAt(int index)
- public int length()
- public String substring(int beginIndex)
- public String substring(int beginIndex, int endIndex)

Java StringBuffer class

String Vs StringBuffer (Programming Example)

```
public class ConcatTest{
   public static String concatWithString()    {
      String t = "ITER";
      for (int i=0; i<10000; i++){
         t = t + "SOADU";
      }
      return t;
   }
   public static String concatWithStringBuffer(){
      StringBuffer sb = new StringBuffer("ITER");
      for (int i=0; i<10000; i++){
         sb.append("SOADU");
      }
      return sb.toString();
   }
```

```
public static void main(String[] args){
      long startTime = System.currentTimeMillis();
      concatWithString();

      System.out.println("Time taken by Concating with
String: "+(System.currentTimeMillis()-startTime)+"ms");

      startTime = System.currentTimeMillis();

      concatWithStringBuffer();

      System.out.println("Time taken by Concating with
 StringBuffer: "+(System.currentTimeMillis()-
startTime)+"ms");
   }
}
```

Java StringBuilder class

- Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.

Constructors:

- StringBuilder()

creates an empty string Builder with the initial capacity of 16

- StringBuilder(String str)

creates a string Builder with the specified string

- StringBuilder(int length)

StringBuilder vs String

String	StringBuilder
• String class is immutable.	• StringBuilder class is mutable
• String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	• StringBuilder is fast and consumes less memory when you cancat strings
• String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method	• StringBuilder class doesn't override the equals() method of Object class

End of Session

Exception Handling

By Dr. Subrat Kumar Nayak
Associate Professor
Department of CSE
ITER, SOADU

Exception Handling in Java

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is Exception in Java?

- Exception is an abnormal condition.
- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

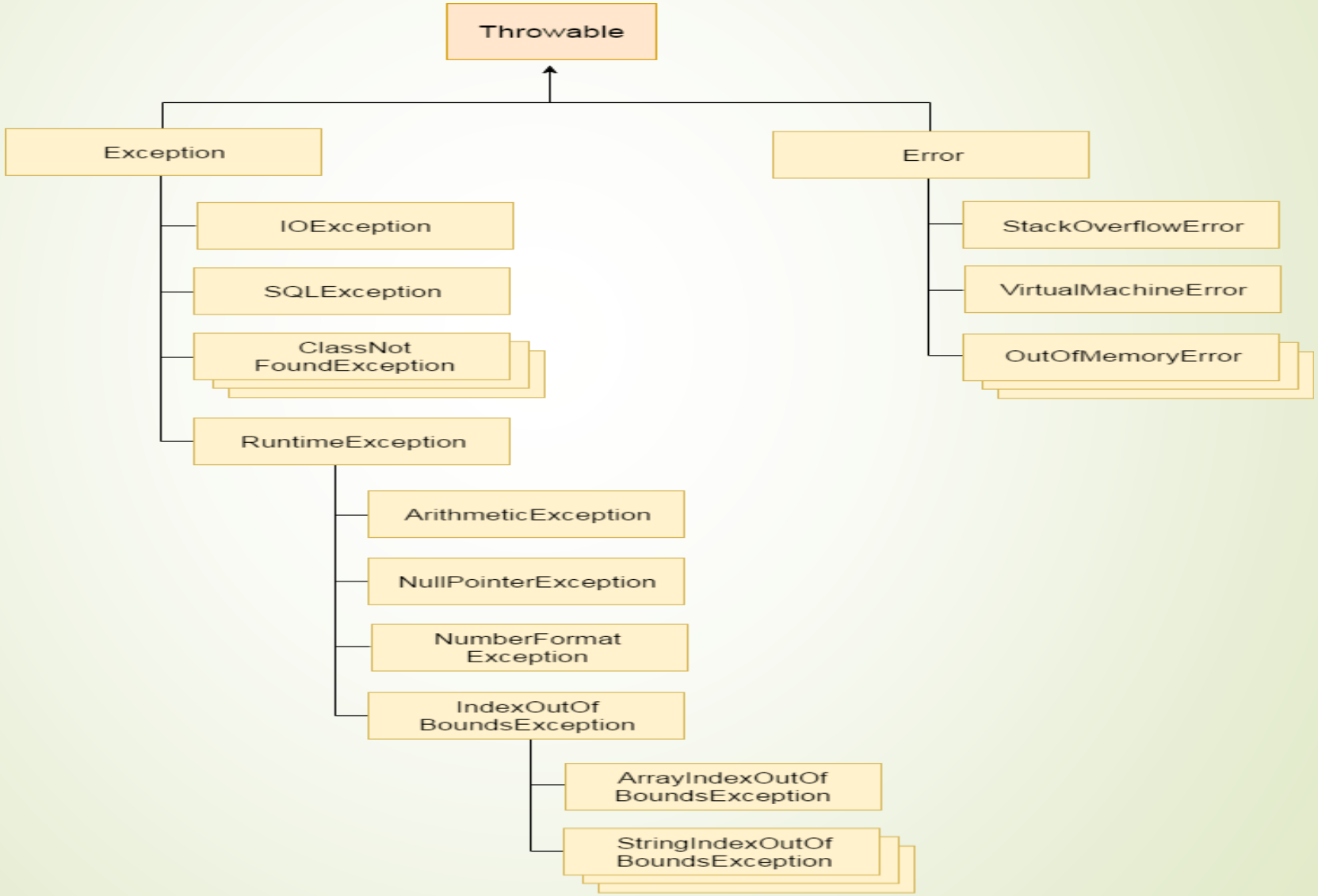
What is Exception Handling?

- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling.

Hierarchy of Java Exception classes



Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception

- The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc.
- Checked exceptions are checked at **compile-time**.

Unchecked Exception

- The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- Unchecked exceptions are not checked at compile-time, but they are checked at **runtime.**

Error

- Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Exception Handling in Java

Example: (Java Exception Handling using a try-catch statement)

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Common Scenarios of Java Exceptions

- A scenario where ArithmeticException occurs

```
int a=50/0;//ArithmeticException
```

- A scenario where NullPointerException occurs

```
String s=null;
System.out.println(s.length());//NullPointerException
```

- A scenario where NumberFormatException occurs

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

- A scenario where ArrayIndexOutOfBoundsException occurs

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Exception Handling in Java

Java try-catch block

try block

- Java **try** block is used to enclose the code that might throw an exception.
- If an exception occurs at the particular statement of try block, the rest of the block code will not execute.
- Java try block must be followed by either catch or finally block.

Syntax:

- **try-catch**

```
try{
    //code that may throw an exception
}catch(Exception_class_Name ref){}
```

- **try-finally**

```
try{
    //code that may throw an exception
}finally{}
```

Exception Handling in Java

catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception (i.e., **Exception**) or the **generated exception** type.
- The catch block must be used after the try block only. You can use **multiple catch block** with a single try block.

Problem without exception handling

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

} }
```

Output: Exception in thread "main" java.lang.ArithmeticException: / by zero

Exception Handling in Java

Solution by exception handling

```
public class TryCatchExample2 {
    public static void main(String[] args) {
        try
        {
              int data=50/0; //may throw exception
        }
        catch(ArithmeticException e)
        {
           System.out.println(e);  / System.out.println("Can't divide by zero");
        }
        System.out.println("rest of the code");
} }
```

Output: java.lang.ArithmeticException: / by zero **Or** Can't divide by zero

rest of the code rest of the code

Exception Handling in Java

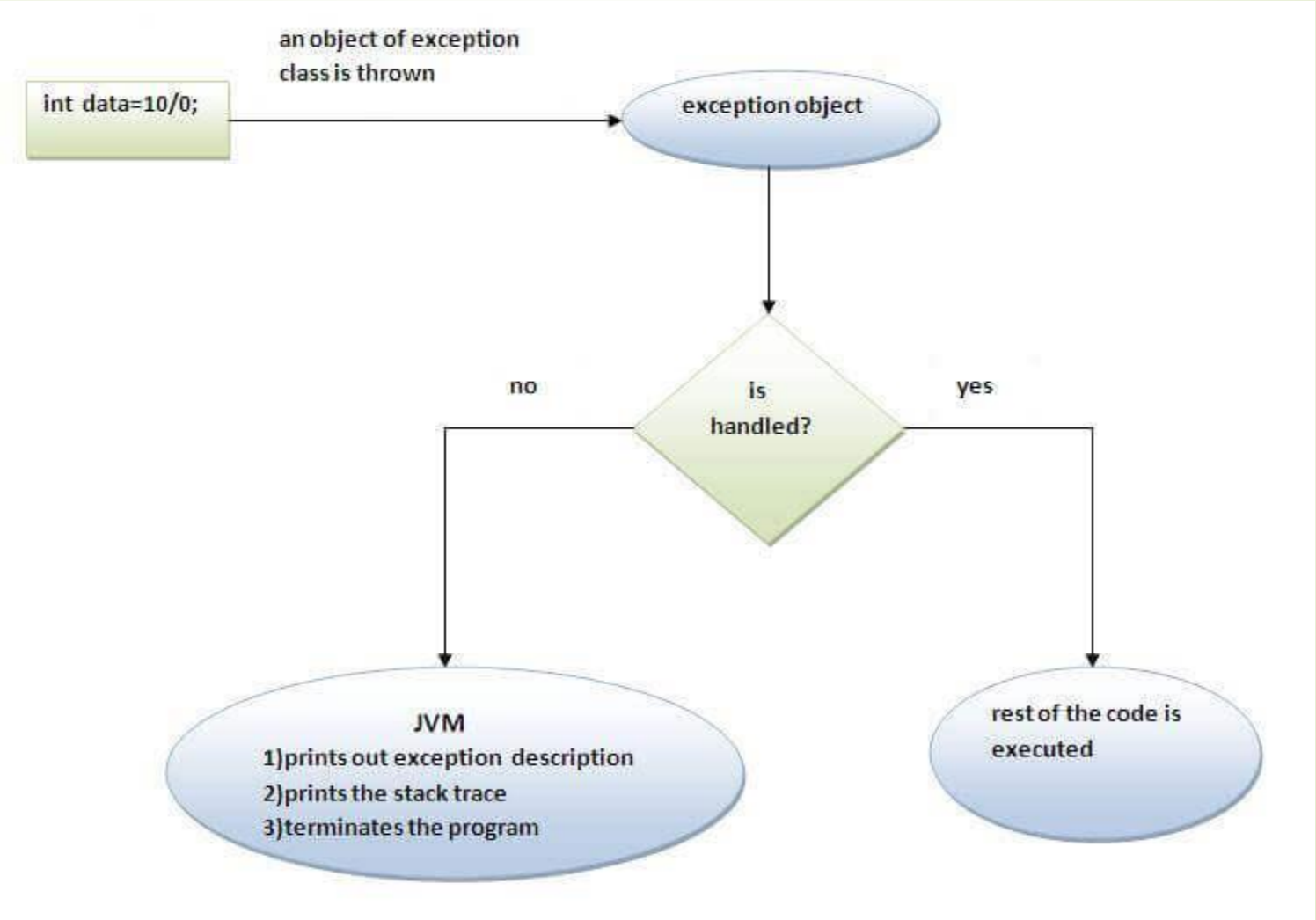
Example: Handling with different type of exception class

```
public class TryCatchExample8 {
    public static void main(String[] args) {
        try
        {
              int data=50/0; //may throw exception
        }
          catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    } }
```

Output: Exception in thread "main" java.lang.ArithmeticException: / by zero

Exception Handling in Java

Internal working of java try-catch block



Exception Handling in Java

Java Multi-catch block

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
        catch(ArithmeticException e)
            {
            System.out.println("Arithmetic Exception occurs");

           }
```

```
catch(ArrayIndexOutOfBoundsException e)
                {
                 System.out.println("ArrayIndexOutOfBounds Exce
ption occurs");
                }
catch(Exception e)
                {
                 System.out.println("Parent Exception occurs");
                }
               System.out.println("rest of the code");
    }
}
```

Exception Handling in Java

Java finally block

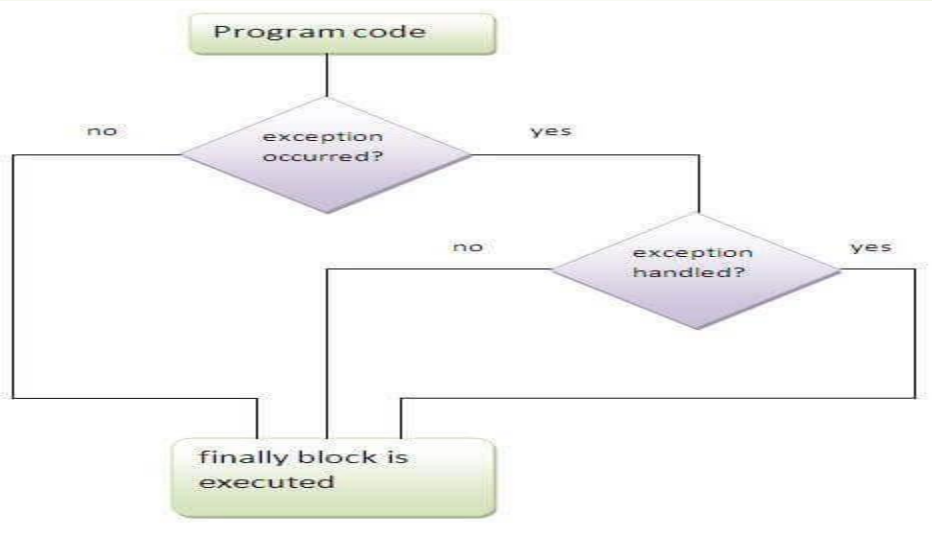
- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
        int data=25/0;
        System.out.println(data);
      }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code..."); //will not execute
  } }
```

Output: finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

Exception Handling in Java



- If you don't handle exception, before terminating the program, JVM executes finally block(if any).
- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Why use java finally?

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Exception Handling in Java

Java throw keyword

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception in java by throw keyword.
- The throw keyword is **mainly used to throw custom exception**.

Syntax

- **throw** exception;

Example

```
public class TestThrow1{
  static void validate(int age){
   if(age<18)
    throw new ArithmeticException("not valid");
   else
    System.out.println("welcome to vote");
  }
  public static void main(String args[]){
    validate(13);
    System.out.println("rest of the code...");
} }
```

Output: Exception in thread main java.lang.ArithmeticException:not valid

Exception Handling in Java

Java Custom Exception

- If you are creating your own Exception that is known as custom exception or user-defined exception.
- Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.

Example:

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
```

```
class TestCustomException1{

  static void validate(int age)throws InvalidAgeException{
    if(age<18)
     throw new InvalidAgeException("not valid");
    else
     System.out.println("welcome to vote");
  }

  public static void main(String args[]){
    try{
    validate(13);
    }catch(Exception m){System.out.println("Exception occured: "+m);}

    System.out.println("rest of the code...");
  }
}
```

Output: Exception occured: InvalidAgeException: not valid

rest of the code...

End of Session

Pattern Matching with Regular Expressions

Dr. Subrat Kumar Nayak
Associate Professor
Department of CSE
ITER, SOADU

Regular Expression

- Regular expressions, or regexes for short, provide a concise and precise specification of patterns to be matched in text.

Example:

- Suppose you have a bunch of 150000 mail in your drive, And let's further suppose that you remember that somewhere in there is an email message from someone named Angie or Anjie. Or was it Angy? But you don't remember what you called it or where you stored it. Obviously, you have to look for it.
- Simplest way is to write a regular expression to search it:

An[^ dn].*

finding words that begin with "An", while the cryptic [^ dn] requires

The "An" to be followed by a character other than (^ means not in this context) a space (to eliminate the very common English word "an" at the start of a sentence) or "d" (to eliminate the common word "and") or "n" (to eliminate Anne, Announcing, etc.).

Subexpression	Matches	Notes
General		
\^	Start of line/string	
$	End of line/string	
\b	Word boundary	
\B	Not a word boundary	
\A	Beginning of entire string	

Subexpression	Matches	Notes
\p{Punct}	Punctuation characters	One of !"#$%&'()*+,-./:;<=>?@[]\^_`{|}\~
\p{Lower}	Lowercase characters	[a-z]
\p{Upper}	Uppercase characters	[A-Z]
\p{XDigit}	Hexadecimal digit characters	[0-9a-fA-F]

Subexpression	Matches	Notes
\z	End of entire string	
\Z	End of entire string (except allowable final line terminator)	See Recipe 4.9
.	Any one character (except line terminator)	
[…]	"Character class"; any one character from those listed	
[\^…]	Any one character not from those listed	See Recipe 4.2
Alternation and Grouping		
(…)	Grouping (capture groups)	See Recipe 4.3
|	Alternation	
(?:_re_)	Noncapturing parenthesis	
\G	End of the previous match	
\ *n*	Back-reference to capture group number "*n*"	
Normal (greedy) quantifiers		
{ *m*,*n* }	Quantifier for "from *m* to *n* repetitions"	See Recipe 4.4
{ *m* ,}	Quantifier for "*m* or more repetitions"	
{ *m* }	Quantifier for "exactly *m* repetitions"	See Recipe 4.10
{,*n* }	Quantifier for 0 up to *n* repetitions	
*	Quantifier for 0 or more repetitions	Short for {0,}
+	Quantifier for 1 or more repetitions	Short for {1,}; see Recipe 4.2
?	Quantifier for 0 or 1 repetitions (i.e., present exactly once, or not at all)	Short for {0,1}
Reluctant (non-greedy) quantifiers		
{ *m*,*n* }?	Reluctant quantifier for "from *m* to *n* repetitions"	
{ *m* ,}?	Reluctant quantifier for "*m* or more repetitions"	
{,*n* }?	Reluctant quantifier for 0 up to *n* repetitions	
*?	Reluctant quantifier: 0 or more	
+?	Reluctant quantifier: 1 or more	See Recipe 4.10
??	Reluctant quantifier: 0 or 1 times	
Possessive (very greedy) quantifiers		
{ *m*,*n* }+	Possessive quantifier for "from *m* to *n* repetitions"	
{ *m* ,}+	Possessive quantifier for "*m* or more repetitions"	
{,*n* }+	Possessive quantifier for 0 up to *n* repetitions	
*+	Possessive quantifier: 0 or more	
++	Possessive quantifier: 1 or more	

Subexpression	Matches	Notes
?+	Possessive quantifier: 0 or 1 times	
Escapes and shorthands		
\	Escape (quote) character: turns most metacharacters off; turns subsequent alphabetic into metacharacters	
\Q	Escape (quote) all characters up to \E	
\E	Ends quoting begun with \Q	
\t	Tab character	
\r	Return (carriage return) character	
\n	Newline character	See Recipe 4.9
\f	Form feed	
\w	Character in a word	Use \w+ for a word; see Recipe 4.10
\W	A nonword character	
\d	Numeric digit	Use \d+ for an integer; see Recipe 4.2
\D	A nondigit character	
\s	Whitespace	Space, tab, etc., as determined by java.lang.Character.isWhitespace()
\S	A nonwhitespace character	See Recipe 4.10
Unicode blocks (representative samples)		
\p{InGreek}	A character in the Greek block	(Simple block)
\P{InGreek}	Any character not in the Greek block	
\p{Lu}	An uppercase letter	(Simple category)
\p{Sc}	A currency symbol	
POSIX-style character classes (defined only for US-ASCII)		
\p{Alnum}	Alphanumeric characters	[A-Za-z0-9]
\p{Alpha}	Alphabetic characters	[A-Za-z]
\p{ASCII}	Any ASCII character	[\x00-\x7F]
\p{Blank}	Space and tab characters	
\p{Space}	Space characters	[\t\n\x0B\f\r]
\p{Cntrl}	Control characters	[\x00-\x1F\x7F]
\p{Digit}	Numeric digit characters	[0-9]
\p{Graph}	Printable and visible characters (not spaces or control characters)	
\p{Print}	Printable characters	

Regular Expression

Differences Among Greedy, Reluctant, and Possessive Quantifiers

- Greedy quantifiers are considered "greedy" because they force the matcher to **read in**, or *eat*, **the entire input string prior to attempting the first match. If the first match attempt** (the entire input string) **fails**, **the matcher backs off the input string by one character and tries again, repeating the process until a match is found or there are no more characters left to back off from.** Depending on the quantifier used in the expression, the last thing it will try matching against is 1 or 0 characters.
- The reluctant quantifiers, however, take the opposite approach: **They start at the beginning of the input string, then reluctantly eat one character at a time looking for a match.** The last thing they try is the entire input string.
- Finally, the possessive quantifiers always eat the entire input string, trying once (and only once) for a match. Unlike the greedy quantifiers, possessive quantifiers never back off, even if doing so would allow the overall match to succeed.

Regular Expression

Differences Among Greedy, Reluctant, and Possessive Quantifiers

Example:

Enter your regex: .*foo // **greedy quantifier**

Enter input string to search: xfooxxxxxx**foo**

I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.

Enter your regex: .*?foo // **reluctant quantifier**

Enter input string to search: **xfoo**xxxxx**xfoo**

I found the text "xfoo" starting at index 0 and ending at index 4.

I found the text "xxxxxxfoo" starting at index 4 and ending at index 13.

Enter your regex: .*+foo // **possessive quantifier**

Enter input string to search: xfooxxxxxx**foo**

No match found. (Failed to backtrack. Hence, **foo** seems to be missing for possesive)

Regular Expression

Greedy	Reluctant	Possessive	Meaning
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times
X+	X+?	X++	X, one or more times
X{n}	X{n}?	X{n}+	X, exactly *n* times
X{n,}	X{n,}?	X{n,}+	X, at least *n* times
X{n,m}	X{n,m}?	X{n,m}+	X, at least *n* but not more than *m* times

Enter your regex: a?
Enter input string to search:
I found the text "" starting at index 0 and ending at index 0.
Enter your regex: a*
Enter input string to search:
I found the text "" starting at index 0 and ending at index 0.
Enter your regex: a+
Enter input string to search:
No match found.

Regular Expression

Q. Write a regular expression to print all the name from n name start with Angie, Anjie or Angy.

Ans: An[^ nd].* //An[^ nd]+ Angelina will not match

Q. Write a regular expression to print the string from bunch of string starting with "A" followed by any number of character.

Ans: A.*/A.+

Q. Write a regular expression to find a match that starts with an alphabate and end with a digit.

Ans: \b\p{Alpha}{1,4}\d{1}\b

Q. Write a regular expression to find a match that starts with a vowel.

Ans: \b[aeiou]\p{Alnum}{1,}

Q. Write a regular expression to find a match that starts with a vowel and end with a vowel.

Ans: \b[aeiou]([0-9] | [a-z]){1,4}[aeiou]\b or

\b[aeiou]\p{Alnum}{1,9}[aeiou]\b or

\b([aeiou] | [AEIOU])\p{Alnum}{1,}[aeiou]\b

Regular Expression

Q. Write a regular expression that matches with a string that starts with a name like Angie/Anjie/Angy.

Ans: ^An[^ dn].*

Q. Write a regular expression that validates a date in MM/DD/YYYY format. Note: ignore leap year

Ans: ^(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}$

Q. Write a regular expression to find a match that starts with an uppercase letter and end with a digit.

Ans: \b\p{Upper}{1}\p{Alpha}{1,}\d{1}\b

Regular Expression

Using regexes in Java: Test for a Pattern

Matching regex using matches() in String class:

- If all you need is to find out whether a given **regex matches a string**, you can use the convenient boolean matches() method of the String class, which accepts a regex pattern in String form as its argument.

Example:

```
if ( inputString . matches ( stringRegexPattern )) {
      // it matched ... do something with it ...
}
```

Regular Expression

Java Regex:

- The **Java Regex** or Regular Expression is an API to *define a pattern for searching or manipulating strings*.

Matching regexes using Pattern and Matcher(s)

- If the regex is going to be used more than once or twice in a program, it is more efficient to construct and use a Pattern and its Matcher (s).
- The normal steps for regex

1. Create a Pattern by calling the static method Pattern.compile().
2. Request a Matcher from the pattern by calling pattern.matcher(CharSequence) for each String (or other CharSequence) you wish to look through.
3. Call (once or more) one of the finder methods (discussed later) in the resulting Matcher .

Regular Expression

java.util.regex package

- The Matcher and Pattern classes provide the facility of Java regular expression.

The Matcher class

It is a *regex engine* which is used to perform match operations on a character sequence.

The Matcher methods

- **boolean matches()**

Used to compare the **entire string** against the pattern; this is the same as the routine in java.lang.String .

- **lookingAt()**

Used to match the pattern **only at the beginning** of the string.

- **boolean find()**

Used to match the pattern in the string (not necessarily at the first character of the string), starting at the beginning of the string or, if the method was previously called and succeeded, at the first character not matched by the previous match.

Regular Expression

Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.

Methods

- static Pattern compile(String regex)

compiles the given regex and returns the instance of the Pattern.

- Matcher matcher(CharSequence input)

creates a matcher that matches the given input with the pattern.

…

Regular Expression

matches()

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = "pqr .*";
                String input ="pqr abd pxy ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. matches ())
                {
                        System .out. println (" Patern "+ pattern +" found in string "+ input );
                }
                else
                {
                        System .out. println (" Patern "+ pattern +" not found in string "+ input );
        }}
}
```

Output : Patern pqr .* found in string pqr abd pxy

Regular Expression

lookingAt()

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = "pqr ";
                String input ="pqr abd pxy ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. lookingAt ())
                {
                        System .out. println (" Patern "+ pattern +" found in string "+ input );
                }
                else
                {
                        System .out. println (" Patern "+ pattern +" not found in string " + input );
        }}
}
```

Output : Patern pqr found in string pqr abd pxy

Regular Expression

find()

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = "abd ";
                String input ="pqr abd pxy ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. find ())
                {
                        System .out. println (" Patern "+ pattern +" found in string "+ input );
                }
                else
                {
                        System .out. println (" Patern "+ pattern +" not found in string "+ input );
        }}
}
```

Output : Patern abd found in string pqr abd pxy

Regular Expression

Finding the Matching Text

- You need to find the text that the regex matched with.

Related functions

start(), end()

- Returns the character position in the string of the starting and ending characters that matched.

groupCount()

- Returns the number of parenthesized capture groups **in the expression/regex**, if any; returns 0 if no groups were used.

group(int i)

- Returns the characters matched by group i of the current match, if i is greater than or equal to zero and less than or equal to the return value of groupCount().
- Group 0 is the entire match, so group(0) (or just group()) returns the entire portion of the input that matched.

Regular Expression

groupCount() example

```
import java . util . regex .*;
      public class RESimple {
            public static void main ( String [] argv ) {
            String pattern = " (.*) (\\d {6}) ";
            //Two groups
            String input =" abdpxy 100000 ";
            Pattern p = Pattern . compile ( pattern );
            Matcher m=p. matcher ( input );
            System .out. println (" Total group = "+m. groupCount ());
      }
}
```

Output : Total group =2

Regular Expression

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = " (.*) (\\d {6}) ";
                String input =" abdpxy 100000 ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. find ()){
                        System .out. println (" Patern "+ pattern +" found in string "+ input +" with group
                                "+m. group (0) );
                                                        // abdpxy 100000
                        System .out. println (" Patern "+ pattern +" found in string "+ input +" with group
                                "+m. group (1) );
                                                        // abdpxy
                        System .out. println (" Patern "+ pattern +" found in string "+ input +" with group
                                "+m. group (2) );
                                                        // 100000
                }
}}
```

Regular Expression

Replacing the Matched Text

- **replaceAll(newString)**

Replaces all occurrences that matched with the new string.

Example:

```
import java . util . regex . Pattern ;
import java . util . regex . Matcher ;
public class ReplaceAll {
        public static void main ( String args [])
        {
                String patt = "\\ bfavor \\b"; // A test input .
                String input = "Do me a favor ? Fetch my favorite .favor ";
                System .out. println (" Input : " + input );
                // Run it from a RE instance and see that it works
                Pattern r = Pattern . compile ( patt );
                Matcher m = r. matcher ( input );
        System .out. println (" ReplaceAll : " + m. replaceAll (" favour "));
}}
```

Regular Expression

- **appendReplacement(StringBuffer, newString)**

Copies up to before the first match, plus the given newString .

- **appendTail(StringBuffer)**

Appends text after the last match (normally used after appendReplacement).

```
public class ReplaceAll {
        public static void main ( String args [])
        {
                String patt = "\\ b favor \\b"; // A test input .
                String input = "Do me a favor ? Fetch my favorite (favor) ";
                System .out. println (" Input : " + input );
                Pattern r = Pattern . compile ( patt ); // Run it from a RE instance and see that it works
                Matcher m = r. matcher ( input );
                StringBuffer sb= new StringBuffer ();
                while (m. find ()) {
                m. appendReplacement (sb , " favour ");// Copy to before first match , plus the word " favor "
                }
                m . appendTail (sb); // copy remainder (comment this line to check the importance of appendTail)
                System .out. println (sb. toString ());
}}
```

Regular Expression

Pattern.compile() Flags

- **CASE_INSENSITIVE**

Turns on case-insensitive matching

Ex. Pattern reCaseInsens = Pattern . compile (pattern, **P**attern . CASE_INSENSITIVE)

// check the previous example using "Favor" instead of "favor".

- **COMMENTS**

Causes whitespace and comments (from # to endofline) to be ignored in the pattern.

- **DOTALL**

Allows dot (.) to match any regular character or the newline, not just any regular character other than newline.

- **MULTILINE**

Specifies multiline mode.

Task (Explore)

- **UNICODE_CASE**
- **UNIX_LINES**

Regular Expression

```
import java . util . regex .*;
public class NewLine
{
        public static void main ( String args [])
        {
                String input = "I dream of engines \ nmore
                engines , all day long ";
                System .out. println (" INPUT : " + input );
                System .out. println ("");
                String [] patt = {" engines . More engines "," ines
                \ nmore "," engines$ "};
                for (int i = 0; i < patt . length ; i ++)
                {
                        System .out. println (" PATTERN " + patt [i]);
                        boolean found ;
                        Pattern p1l = Pattern . compile ( patt [i]);
found = p1l. matcher ( input ). find ();
System .out. println (" DEFAULT match " + found );
Pattern pml =
                                Pattern . compile ( patt [i], Pattern .
                                DOTALL | Pattern . MULTILINE );
found = pml. matcher ( input ). find ();
System .out. println (" MultiLine match " + found );
System .out. println ();
}}}
```

Output:

```
INPUT : I dream of engines
more engines , all day long
PATTERN engines . more engines
DEFAULT match false
MultiLine match true
PATTERN ines
more
DEFAULT match true
MultiLine match true
PATTERN engines$
DEFAULT match false
MultiLine match true
```

End of Chapter

Numbers

(Wrapper Classes in Java)

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Wrapper Classes in Java

Wrapper class:

- The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Why Wrapper class?

- Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc.

1. Change the value in Method:

Java supports only call by value. So, if we pass a primitive value, it will not change the original value.

2. Serialization:

We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.

3. Synchronization:

Java synchronization works with objects in Multithreading.

4. java.util package:

The java.util package provides the utility classes to deal with objects.

5. Collection Framework:

Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Wrapper Classes in Java

- The list of primitive types and their corresponding Wrapper classes.

Built-in type	Object wrapper	Size of built-in (bits)	Contents
byte	Byte	8	Signed integer
short	Short	16	Signed integer
int	Integer	32	Signed integer
long	Long	64	Signed integer
float	Float	32	IEEE-754 floating point
double	Double	64	IEEE-754 floating point
char	Character	16	Unsigned Unicode character
n/a	BigInteger	unlimited	Arbitrary-size immutable integer value
n/a	BigDecimal	unlimited	Arbitrary-size-and-precision immutable floating-point value

- Along with these, there is another wrapper class for boolean primitive data type, i.e. Boolean wrapper class
- The **java.lang.Number** class is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.The Subclasses of Number must provide methods to convert the represented numeric value to byte, double, float, int, long, and short.

Wrapper Classes in Java

Constructors:

- Every wrapper class in java has two constructors,

(a) First constructor takes corresponding primitive data as an argument

(b) Second constructor takes string as an argument.

Notes:

- The string passed to second constructor should be parse-able to number , otherwise you will get run time **NumberFormatException**.
- Wrapper Class Character has only one constructor which takes char type as an argument. It doesn't have a constructor which takes String as an argument. Because, String can not be converted into Character.
- Wrapper class Float has three constructors. The third constructor takes double type as an argument.

Examples:

```
Integer I1 = new  Integer(30);
//Constructor which takes int value as an argument

Integer I2 = new  Integer("30");
//Constructor which takes String as an argument
```

Wrapper Classes in Java

Examples…

```
Short S1 = new Short((short) 20);   //Constructor which takes short value as an argument
Short S2 = new Short("10");         //Constructor which takes String as an argument


Float F1 = new Float(12.2f); //Constructor which takes float value as an argument
Float F2 = new Float("15.6");//Constructor which takes String as an argument
Float F3 = new Float(15.6d); //Constructor which takes double value as an argument


Character C1 = new Character('D'); //Constructor which takes char value as an argument
Character C2 = new Character("a");
//Compile time error : String a can not be converted to character
```

- Converting primitive data types into object is called **boxing**, and this is taken care by the compiler.
- Similarly, the Wrapper object can also be converted back to a primitive data type, and this process is called **unboxing**.

- The **automatic conversion** of primitive data type into its corresponding wrapper class is known as **auto-boxing**.
- The **automatic conversion** of wrapper type into its corresponding primitive type is known as **auto-unboxing**.

Wrapper Classes in Java

Examples for Auto-boxing & Auto-unboxing:

```
public class WrapperExample1{
public static void main(String args[]){
int a=20;
Integer j=a;//auto-boxing, now compiler will call Integer.valueOf(a) internally
System.out.println(j);// will print 20
}}
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(10);
int j=a;//auto-unboxing, now compiler will call a.intValue() internally
System.out.println(j);// will print 10
}}
```

Wrapper Classes in Java

Number Methods:

Following is the list of the instance methods that **all the subclasses** of the Number class implements.

xxxValue()

Converts the value of *this* Number object to the **xxx** data type and returns it.

Syntax:

xxx xxxValue()

- xxx stands for different primitive data type.

Example:

```
public class Test {
public static void main(String args[]) {
     Integer x = 5;
     // Returns byte primitive data type
     System.out.println(x.byteValue());
     // Returns double primitive data type
     System.out.println(x.doubleValue());
     // Returns long primitive data type
     System.out.println(x.longValue());
     //will not work x.charValue()for Integer.
} }
```

```
Output:
5
5.0
5
```

Wrapper Classes in Java

compareTo()

- The method compares the Number object that invoked the method to the argument. It is possible to compare Byte, Long, Integer, etc.
- However, two different types cannot be compared, both the argument and the Number object invoking the method should be of the same type.

Syntax:

```
public int compareTo(NumberSubClass referenceName)
```

Return Value:

- If this object equal to the argument then 0 is returned.
- If this object less than the argument then -1 is returned.
- If this object greater than the argument then 1 is returned.

Example:

```
public class Test {
public static void main(String args[]) {
     Integer x = 5;
     System.out.println(x.compareTo(3));
     System.out.println(x.compareTo(5));
     System.out.println(x.compareTo(8));
} }
```

```
Output:
1
0
-1
```

Wrapper Classes in Java

valueOf()

- The valueOf method returns the relevant Number Object holding the value of the argument passed. The argument can be a primitive data type, String, etc.
- This method is a **static** method. The method can take two arguments, where one is a String and the other is a radix.

Syntax:

```
static Xxx valueOf(xxx i)// works for all Wrapper type

//xxx: primitive type of corresponding wrapper type

static Xxx valueOf(String s)// does not work for Character

static Xxx valueOf(String s, int radix)// not applicable for Character,
Double,Float.

//Xxx represents any of the subclass of Number class(Integer, Float etc.)
```

Return Values:

- **valueOf(int i)** – This returns an Xxx object holding the value of the specified primitive.
- **valueOf(String s)** – This returns an Xxx object holding the value of the specified string representation.
- **valueOf(String s, int radix)** – This returns an Xxx object holding the integer value of the specified string representation, parsed with the value of radix.

Wrapper Classes in Java

Example:

```
public class Test {
public static void main(String args[]) {
    Integer x =Integer.valueOf(9);
    Double c = Double.valueOf(5);
    Float a = Float.valueOf("80");
    Integer b = Integer.valueOf(“0111",2);
    System.out.println(x);
    System.out.println(c);
    System.out.println(a);
    System.out.println(b);
} }
```

```
Output:
9
5.0
80.0
7
```

End of Session

Numbers

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Some More Functions in Wrapper Classes…

equals()

- The method determines whether the Number object that invokes the method is equal to the object that is passed as an argument.

Syntax

```
public boolean equals(Object o)
```

Return Value:

- The method returns True if the argument is not null and is an object of the same type and with the same numeric value.

Example:

```
public class Test {
public static void main(String args[]) {
        Integer x = 5;
        Integer y = 10;
        Integer z =5;
        System.out.println(x.equals(y));
        System.out.println(x.equals(z));
} }
```

Some More Functions in Wrapper Classes…

toString()

- The method is used to get a String object representing the value of the Number Object.
- If the method takes a primitive data type as an argument, then the String object representing the primitive data type value is returned.
- If the method takes two arguments, then a String representation of the first argument in the radix specified by the second argument will be returned.

Syntax:

```
String toString()
static String toString(int i)
Static String toString(int i, int radix)
```

Example :

```
public class Test {
public static void main(String args[]) {
Integer x = 5;
System.out.println(x.toString());
System.out.println(Integer.toString(12));
System.out.println(Integer.toString(12,2));
} }
```

```
Output:
5
12
1100
```

Some More Functions in Wrapper Classes…

parseXxx()

- This method is used to get the primitive data type of a certain String. parseXxx() is a static method and can have one argument or two.
- Does not work for Character.

Note: same as valueOf(). However, it only works on String object.

Syntax:

```
static int parseInt(String s)
static int parseInt(String s, int radix)
```

Example:

```
public class Test { public static void main(String args[]) {
    int x =Integer.parseInt("9");
    double c = Double.parseDouble("5");
    int b = Integer.parseInt("1100",2);
    System.out.println(x);
    System.out.println(c);
    System.out.println(b);
} }
```

```
Output:
9
5
12
```

How much we have proceeded?

- **Introduction**
- **Checking Whether a String Is a Valid Number (parseInt())**
- **Wrapper Class and its Methods**
- **Converting Numbers to Objects and Vice Versa**

Storing a Larger Number in a Smaller Number

```
float f=3.0// won't even compile!
This line will be understood as follows.
double tmp=3.0;
float f=tmp;
```

How to fix?

- Can be fixed in one of the several ways:

1) By making the 3.0 a float (probably the best solution)
2) By making f a double
3) By putting in a cast
4) By assigning an integer value of 3, which will get "promoted"

Example:

```
float f=3.0f;
double f=3.0;
float f=(float)3.0;
float f=3;
```

Ensuring the Accuracy of Floating-Point Numbers

- In java integer division by 0 consider as logical error so it throws an ArithmeticException.
- Floating-point operations, however, do not throw an exception because they are defined over an (almost) infinite range of values.

Java act differently for the following cases.

1) Java signal errors by producing the constant POSITIVE_INFINITY if you divide a positive floating-point number by zero

2) It signal constant NEGATIVE_INFINITY if you divide a negative floating-point value by zero.

3) Produces NaN (Not a Number) if you otherwise generate an invalid result

- Values for these **three public constants** are defined in both the Float and the Double wrapper classes.
- The value NaN has the unusual property that it is not equal to itself (i.e., NaN != NaN).
- x==NaN never be true, instead, the methods Float.isNaN(float) and Double.isNaN(double) must be used.

Ensuring the Accuracy of Floating-Point Numbers

```
public static void main(String[] args){
    double d = 123;
    double e = 0;
    if (d/e == Double.POSITIVE_INFINITY)
        System.out.println("Check for POSITIVE_INFINITY works");
    double s = Math.sqrt(-1);
    if (s == Double.NaN)
        System.out.println("Comparison with NaN incorrectly returns
        true");
    if (Double.isNaN(s))
        System.out.println("Double.isNaN() correctly returns true");
}
```

Output:

```
Check for POSITIVE_INFINITY works
Double.isNaN() correctly returns true
```

Comparing Floating Point Numbers

- The equals() method of Float and Double wrapper class returns true if the two values are the same bit for bit (i.e., if and only if the numbers are the same or are both NaN).
- It returns false otherwise, including if the argument passed in is null, or if one object is +0.0 and the other is –0.0.
- To actually compare floating-point numbers for equality, it is generally desirable to compare them within **some tiny range of allowable differences**; this range is often regarded as a tolerance or as *epsilon*.

Example:

```
public class NumberTest {
    public static void main(String[] args) {
        float x=0.3f*3;
        if(x==0.9)
            System.out.println(x);
    }
}
```

Output:

Comparing Floating Point Numbers

```
public class FloatCmp {
   final static double EPSILON = 0.0000001;
   public static void main(String[] argv) {
      double da = 3 * .3333333333;
      double db = 0.99999992857;
      // Compare two numbers that are expected
      to be close.
      if (da == db) {
        System.out.println("Java considers " +
        da + "==" + db);
       // else compare with our own equals
       overload
      }
      else if (equals(da, db, 0.0000001)) {
        System.out.println("Equal within epsilon
        " + EPSILON);
      }
      else {
        System.out.println(da + " != " + db);
      }
}
```

```
/** Compare two doubles within a given
epsilon */
public static boolean equals(double a, double
b, double eps) {
   if (a==b)
      return true;
   // If the difference is less than epsilon,
   treat as equal.
   return Math.abs(a - b) < eps;
}
/** Compare two doubles, using default
epsilon */
public static boolean equals(double a, double
b) {
   return equals(a, b, EPSILON);
}
```

Rounding Floating-Point Numbers

- To round floating-point numbers properly, use Math.round() .
- It has two overloads:

  if you give it a double , you get a long result;

  if you give it a float , you get an int .

- If the argument is *NaN* (not a number), then the function will return 00.
- If the argument is negative infinity (**Float**) or any value less than or equal to the value of **Integer.MIN_VALUE**, then the function returns **Integer.MIN_VALUE**. (Try for Double.NEGATIVE_INFINITY)
- If the argument is positive infinity or any value greater than or equal to the value of Integer.MAX_VALUE, then the function returns Integer.MAX_VALUE. (Try for Double.POSITIVE_INFINITY)

Example:

```
double d=5.67;
System.out.println(Math.round(d));
float f=9.4255f;
System.out.println(Math.round(f));
```

End of Session

Numbers

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Formatting Numbers

- **NumberFormat** is an **abstract base class** for all number formats. This class provides the interface for formatting and parsing numbers.
- NumberFormat also provides methods for determining which locales (US, India, Italy, etc) have number formats, and what their names are.
- NumberFormat helps you to format and parse numbers for any locale.
- A DecimalFormat object appropriate to the user's locale can be obtained from the factory method NumberFormat.getInstance() and manipulated using set methods.

Package :

`Java.text.NumberFormat`

How to Create a NumberFormat object?

NumberFormat nf=NumberFormat.getInstance()// No constructor

Methods:

setMaximumFractionDigits()

- Sets the maximum number of digits allowed in the fraction portion of a number.

Syntax:

void setMaximumFractionDigits(int newValue)

Formatting Numbers

Example:

```
import java.text.NumberFormat;
public class NumberFormatTest {
    public static void main(String args[])
    {
        NumberFormat nf=NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        double d=123.456;
        System.out.println(nf.format(d));// format the value with format()
    }
}
```

Output:

```
123.46
```

Formatting Numbers

void setMaximumIntegerDigits(int newValue)

- Sets the maximum number of digits allowed in the integer portion of a number.

void setMinimumFractionDigits(int newValue)

- Sets the minimum number of digits allowed in the fraction portion of a number.

void setMinimumIntegerDigits(int newValue)

- Sets the minimum number of digits allowed in the integer portion of a number.

Example:

Q. Write a program to set the minimum integer digit to 3, maximum fraction digit to 4 and minimum fraction digit to 2 of a decimal number.

Formatting Numbers

Changing the pattern dynamically

- You can also construct a DecimalFormat with a particular pattern or change the pattern dynamically using applyPattern().

Common Pattern Characters

Character	Meaning
#	A digit, leading zeroes are omitted.
0	A digit - always displayed, even if number has less digits (then 0 is displayed)
.	Locale-specific decimal separator (decimal point)
,	Locale-specific grouping separator (comma in English)
-	Locale-specific negative indicator (minus sign)
%	Shows the value as a percentage
;	Separates two formats: the first for positive and the second for negative values
'	Escapes one of the above characters so it appears
Anything else	Appears as itself

Formatting Numbers

Examples:

```
import java.text.NumberFormat;
import java.text.DecimalFormat;
public class NumberFormatTest {
     public static void main(String args[])
     {
          NumberFormat ourForm = new DecimalFormat("###.##");
          double d=123.345;
          System.out.println(ourForm.format(d));
     }
}
```

Output:

```
123.34
```

Formatting Numbers

```
import java.text.NumberFormat;
import java.text.DecimalFormat;
public class NumberFormatTest {
    public static void main(String args[])
    {
        NumberFormat ourForm = new DecimalFormat("0000.##");
        double d=12.5678;
        System.out.println(ourForm.format(d));
    }
}
```

Output:

0012.57

Converting Between Binary, Octal, Decimal, and Hexadecimal

- Integer.toString(int input, int radix) to convert from integer to any type.

Example

```
public class ConversionTest {
    public static void main(String args[])
    {
        int i=42;
        String res1=Integer.toString(i,2);
        String res2=Integer.toString(i,8);
        String res3=Integer.toString(i,16);
        String res4=Integer.toString(i,10);
        System.out.println("42 in base 2 is "+res1);
        System.out.println("42 in base 8 is "+res2);
        System.out.println("42 in base 16 is "+res3);
        System.out.println("42 in base 10 is "+res4);
    }
}
```

Output:
42 in base 2 is 101010
42 in base 8 is 52
42 in base 16 is 2a
42 in base 10 is 42

Converting Between Binary, Octal, Decimal, and Hexadecimal

- Integer.parseInt(String input, int radix) to convert from any type of number to an Integer

```
public class ConversionTest {
public static void main(String args[])
{
    String str="1010";
    Integer iObj=Integer.parseInt(str,2);
    System.out.println("1010 in base 2 is "+iObj);
    Integer iObj1=Integer.parseInt(str,8);
    System.out.println("1010 in base 8 is "+iObj1);
    Integer iObj2=Integer.parseInt(str,16);
    System.out.println("1010 in base 16 is "+iObj2);
}
}
```

Output:
1010 in base 2 is 10
1010 in base 8 is 520
1010 in base 16 is 4112

End of Session

Numbers

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Operating on a Series of Integers

- For a contiguous set, use a for loop.

Example:

```
String months[]={
"January", "February", "March", "April",
"May", "June", "July", "August",
"September", "October", "November", "December"
};
for(int i=0;i<months.length;i++)
{
System.out.println("Month " + months[i])
}
```

Operating on a Series of Integers

- For discontinuous ranges of numbers, use a java.util.BitSet .

```
// A discontiguous set of integers, using a BitSet
// Create a BitSet and turn on a couple of bits.
String months[]={"January", "February", "March", "April", "May", "June", "July",
"August", "September", "October", "November", "December‟ };
BitSet b = new BitSet();
b.set(0);
// January
b.set(3);
// April
b.set(8);
// September
// Presumably this would be somewhere else in the code.
for (int i = 0; i<months.length; i++)
{
    if (b.get(i))
    System.out.println("Month " + months[i]);
}
```

Output:
Month January
Month April
Month September

Generating Random Numbers

- Use java.lang.Math.random() to generate random numbers.

Example:

```
//java.lang.Math.random( ) is static, don't need any constructor calls
System.out.println("A random from java.lang.Math is " + Math.random( ));
```

- If you need integers random number, construct a java.util.Random object and call its nextInt() method.

Example:

```
public class RandomInt {
public static void main(String[] a) {
Random r = new Random();
for (int i=0; i<1000; i++)
{
    // nextInt(10) goes from 0-9; add 1 for 1-10;
    System.out.println(1+r.nextInt(10));
}}}
```

Generating Random Numbers

Some other nextXxx()methods from java.util.Random class

- boolean nextBoolean() to find a random boolean .
- byte nextByte() to find a random boolean .
- int nextInt() to find a random int.
- float nextFloat() to find a random float.
- double nextDouble() to find next double.

Handling Very Large Numbers

- **Can you store 100! using any primitive data type?**
- In order to handle very large numbers java provides two classes from java.math package.

(1) BigInteger, to create a large integer number.

(2) BigDecimal, to create a large decimal number(Real number).

BigInteger

- BigInteger class is used for mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types.

Constructors:

- `BigInteger(String val)`

Translates the decimal String representation of a BigInteger into a BigInteger.

- `BigInteger(String val, int radix)`

Translates the String representation of a BigInteger in the specified radix into a BigInteger.

Handling Very Large Numbers

Example:

```
import java.math.BigInteger;
public class BigIntTest {
        public static void main(String args[])
        {
                BigInteger bi=new BigInteger("1234455666666");
                System.out.println(bi);
                BigInteger bi1=new BigInteger("1111",2);
                System.out.println(bi1);
        }
}
```

```
Output:
1234455666666
15
```

Handling Very Large Numbers

Methods:

abs()

- It returns a BigInteger, whose value is the absolute value of this BigInteger.

Example:

```
import java.math.BigInteger;
public class BigIntegerAbsExample {
public static void main(String[] args){
BigInteger big1, big2, big3, big4;  // create 4 BigInteger objects
big1=new BigInteger("345");// assign value to big1
big2=new BigInteger("-345");    // assign value to big2
big3=big1.abs ();         // assign absolute value of big1 to big 3
big4=big2.abs ();        // assign absolute value of big 2 to big 4.
String str1 = "Absolute value of" + big1 + "is" + big3;
String str2 = "Absolute value of" + big2 + "is" + big4;
System.out.println(str1);
System.out.println(str2 );
} }
```

Handling Very Large Numbers

add()

- This method returns a BigInteger by simply computing 'this + val' value.

```
public class BigIntTest {
public static void main(String args[])
{
    BigInteger bi1=new BigInteger("1234455666456");
    BigInteger bi2=new BigInteger("1234455666456");
    BigInteger bi3=bi1.add(bi2);
    System.out.println("Sum of big integer= "+bi3);
}}}
```

```
Output:
Sum of big integer= 2468911332912
```

Handling Very Large Numbers

`BigInteger multiply(BigInteger val)`

- Returns a BigInteger whose value is (this * val).

`BigInteger divide(BigInteger val)`

- Returns a BigInteger whose value is (this / val).

`int compareTo(BigInteger val)`

- Compares this BigInteger with the specified BigInteger.

`boolean equals(Object x)`

- Compares this BigInteger with the specified Object for equality.

`float floatValue()`

- Converts this BigInteger to a float.

`int intValue()`

- Converts this BigInteger to a int.

Handling Very Large Numbers

BigDecimal()

BigDecimal class is used for mathematical operation which involves very big real number calculations that are outside the limit of all available primitive data types.

Constructors:

- `BigDecimal(String val)`

Translates the string representation of a BigDecimal into a BigDecimal object.

- `BigDecimal(BigInteger val)`

Translates a BigInteger into a BigDecimal.

Handling Very Large Numbers

Methods of BigDecimal

`BigDecimal abs()`

- Returns a BigDecimal whose value is the absolute value of this BigDecimal.

`BigDecimal add(BigDecimal augend)`

- Returns a BigDecimal whose value is (this + augend).

`BigDecimal divide(BigDecimal divisor)`

- Returns a BigDecimal whose value is (this / divisor).

`BigDecimal multiply(BigDecimal multiplicand)`

- Returns a BigDecimal whose value is (this * multiplicand). + multiplicand.scale()).

`int compareTo(BigDecimal val)`

- Compares this BigDecimal with the specified BigDecimal.

`boolean equals(Object x)`

- Compares this BigDecimal with the specified Object for equality.

End of Session